ESD-TDR-64-108 SR-99

FORSIM IV

FORTRAN IV SIMULATION LANGUAGE USER'S GUIDE

TECHNICAL DOCUMENTARY REPORT NO. ESD-TDR-64-108

MAY 1964

E. Famolari

Prepared for

416 L/M-CONTROL-WARNING SUPPORT SYSTEM

ELECTRONIC SYSTEMS DIVISION

AIR FORCE SYSTEMS COMMAND

UNITED STATES AIR FORCE

L. G. Hanscom Field, Bedford, Massachusetts

Project 416. 2

Prepared by

THE MITRE CORPORATION
Bedford, Massachusetts

Contract AF 19(628)-2390

ESD-TDR-64-108 SR-99

FORSIM IV

FORTRAN IV SIMULATION LANGUAGE USER'S GUIDE

TECHNICAL DOCUMENTARY REPORT NO. ESD-TDR-64-108

MAY 1964

E. Famolari

Prepared for

416 L/M-CONTROL-WARNING SUPPORT SYSTEM
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Project 416. 2
Prepared by

THE MITRE CORPORATION
Bedford, Massachusetts

Contract AF 19(628)-2390

FOREWORD

FORSIM IV was developed by the author as an aid to simulating certain aspects of the Back-Up Interceptor Control (BUIC) system. While the motive force was a specific application, FORSIM IV is the outgrowth of a study of various simulation languages and techniques. As a result, the author believes that it has potentially wider applications as a general simulation pseudo-language. It is the first language of this type developed for the IBM 7030 computer. (By assembly of FORSIM IV using existing FAP coded routines in place of STRAP coded routines, it also could be used on any FORTRAN IV compiling 7090 computer.)

Insofar as can be determined, it is also the first instance of a working, subroutine-structured simulation language. The simple but effective technique of subroutine structure primarily results from the flexibility of the IBM FORTRAN IV language. The following are the more important of the many benefits which are derived from employing a routine structure: a common or "machine-independent" structure, providing ease of conversion and expansion; elimination of a costly precompiler and a new operating system; and simplification of system structure and maintenance. Furthermore, FORSIM IV can be integrated into existing FORTRAN oriented program packages (Loaders, etc.).

The simulation facilities of MITRE include the FAST precompiler language and the STAPP System which can be combined with FORSIM to ease the work of preparing large simulations by subdividing it among several programmers.

The author is indebted to W. L. Cleveland for providing the debug and random number routines.

FORSIM IV---FORTRAN IV SIMULATION LANGUAGE USER'S GUIDE

ABSTRACT

FORSIM IV was developed as an aid to simulating certain aspects of the Back-Up Interceptor Control (BUIC) system, and is the first general simulation pseudo-language developed for the IBM 7030 computer. It represents an innovation in simulation language technique since it is constructed not as a language, but as a subroutine package. It can be adapted to any computer capable of compiling programs written in FORTRAN IV language. Constructed in FORTRAN IV, FORSIM IV is conceptually related to Control and Simulation Language (CSL); however, it provides commands and services not available in CSL, while its subroutine structure provides for the easy expansion of the command set, as well as virtual machine independence. A model FORSIM IV program is included in the document.

REVIEW AND APPROVAL

Publication of this technical documentary report does not constitute Air Force approval of the report's findings or conclusions. It is published only for the exchange and stimulation of ideas.

[illegible]

F. R. EDGIN
Chief, Engineering Division
416L/M System Program Office

CONTENTS

SECTION I
INTRODUCTION

FORSIM IV is a general purpose simulation package related to FORTRAN IV. It represents an innovation in simulation language technique, since it is constructed not as a language (i. e., a precompiler) but as a subroutine package. Presently it is available for use on the IBM 7030 (STRETCH) computer. With a small amount of effort, it can be adapted to any computer capable of compiling programs written in FORTRAN IV language. This is a result of the structure of FORSIM IV. It is a package, that is, a collection of subroutines; all of the subroutines, with the exception of the debug feature and the random number generators, are written in FORTRAN IV.

To the user, the difference between a simulation language and FORSIM IV is that in using a language he must write, "Wonderful Statement," whereas, in using FORSIM IV, he must write, "Call Wonderful Subroutine." Furthermore, the user of FORSIM IV must write a FORTRAN IV program, calling subroutines as he needs them. This should not be viewed as a disadvantage, since any really flexible simulation language, no matter how self-contained, requires the services of a professional programmer.

Many languages consist of "Wonderful Statements" and "Ordinary Statements." The "Wonderful Statements" provide the power and flexibility of the language. The "Ordinary Statements" are generally equivalent to simple FORTRAN statements. Such a language (e.g., SIMSCRIPT or CSL) is usually precompiled into FORTRAN. Thus, the "Wonderful Statement" becomes a sequence of FORTRAN statements. Clearly, if one can transform the "Wonderful Statements" into "Wonderful Subroutines" then one has constructed a FORTRAN package whose level is effectively higher than FORTRAN language.

In this sense, FORSIM IV is a transformation of CSL (Control and Simulation Language), a joint product of Laski, Esso Petroleum (U.K.), and Buxton, IBM (U.K.). Full credit is given Dr. Laski for the conceptual framework of FORSIM IV. CSL was chosen because it is flexible, powerful, and easy to learn.

FORSIM should not be viewed simply as a transliteration of CSL. It provides commands and services not available in CSL, while its subroutine structure provides for the easy expansion of the command set, as well as virtual machine independence. Furthermore, since so much of a FORSIM program must be written in FORTRAN, one retains the advantage of familiarity, while gaining the effect of a versatile simulation language.

SECTION II
FORSIM IV

BUILDING BLOCKS

Concept

FORSIM IV is founded upon the concepts of Entity, Class, and Set. An Entity is the basic unit of one's simulation, e.g., a specific truck if a trucking line is being simulated, or a specific message if a communication network is being simulated. An Entity is something which flows through the simulated system; it is similar to a transaction in the Gordan Simulator.

Whereas an Entity is a specific unit, a Class is the grouping of all Entities of the same type, e.g., the Class of All Trucks or of All Messages. Sets are used to describe relations between Entities of the same Class. One might have a Set of All Trucks which are loaded, and another Set of All Trucks which are garaged for repairs. Sets are always defined relative to a specific Class of Entities. Thus, given a Class of Trucks and a Class of Drivers, one could ascribe to SETA all Drivers on the road and to SETB all Trucks on the road, but Drivers could not be included in SETB nor Trucks in SETA. Sets serve, in the Gordon Simulator sense, as Stores, Facilities, and Queues. Sets may overlap; i.e., they may contain some common Entities.

Suppose we have a Class of Trucks and that for each Truck (Entity), we know the load capacity and the operating cost per mile. These bits of information are called Attributes, or Parameters of the Entities. Attributes are associated with each Entity of a Class. Entities may have no Attributes or many, subject only to core storage limitations.

We may form queues of Entities. Queues may be LIFO (Last In, First Out), FIFO (First In, First Out), Random, or Ranked. A Queue contains

Entities in some order. Thus, a Queue is a Set. The ordering depends on how we use the Set. All Sets are ordered collections of Entities, and the ordering is automatically preserved as Entities are inserted or removed from the Set.

Consider SETA which contains Truck 1 and Truck 2, and suppose that we desire to insert Truck 3. If SETA is ordered Last (1, 2) First, we insert Truck 3 and obtain Last (3, 1, 2) First. To treat SETA as a FIFO Queue, we command that the first Entity (Truck 2) be removed when we require a truck. If SETA is a LIFO Queue, we command that the last Entity (Truck 3) be removed. Alternatively, we could insert Entities at the head of SETA. Thus, SETA would become, after inserting Truck 3, Last (1, 2, 3) First. Then we could treat SETA as a LIFO Queue by always removing the first Entity.

Random Queues are obtained by normal insertion of Entities into a Set and removal of Entities via the command subroutine, ANY, which picks some Entity at random from all the Entities in the Set. Also we may form a Set and, then by using the command subroutine, RANK, reorder it according to the values of some specific Attribute of the Entities in the Set. After RANK is employed, the Entity with the maximum Attribute value is first in the Set, and the Entity with the minimum Attribute value is last. One may then remove the first or last Entity in the Set. Thus, one treats the Set as a Ranked Queue.

To perform all of the varied actions and tests associated with the relations between Set, Entities, and Attributes, a large number of commands have been formulated. These commands, or subroutines, are called Set-Entity Relations and are described in detail in Section III.

Structure

FORSIM IV consists of 50 subroutines, which are somewhat more, rather than less independent. They are held together by the concept of Entity-Set

Relations. For example, consider a Class of 100 Entities which may occupy any of 3 Sets (SETA, SETB, SETC). Since Entities fill Sets, there must be 100 slots to each set.*

To allow for Entities to be time ordered (or otherwise ordered), we treat each Set as a pushdown list. Thus, each Set in the core is a table of 101 slots. The first slot contains a count of the number of slots currently in use (never less than 1, since it counts itself). The second slot contains the most recent entry, etc. An Entity is represented in these slots by its index, which in our example can be from 1 to 100. Thus, one could visualize a Class of Entities as an indexed table, with each Entity represented by a single index value, and a Set as a push-down list containing index values or Entities.

Actually, it is not necessary to define a core table for a Class, since the conceptual manipulation of Entities reduces, in the computer, to the actual manipulation of index values. However, if we wish to associate Attributes (i.e., Parameters) or time values with each Entity, then we must create appropriately dimensioned core tables. These core tables are referenced by index values in Sets. Thus, if we have 10 as the first index value in SETA and if we desire to set the fifth of its 10 Attributes equal to 5 (assume ATRB (100, 10)), then we write:

```
(1)    CALL FIRST (I, SETA, LOG)
(2)    ATRB (I, 5) = 5
```

I is set to 10, the first Entity in SETA, by line (1); then line (2) causes the fifth Attribute of the Ith Entity to be set to 5.

---

*N. B. If SETC can contain only 1 Entity at a time, we can specify that it have only 1 slot and not 100.

Thus, one must properly create and dimension all Sets, Attribute Tables, or Time Tables. Then, bearing in mind the conceptual Entity-Set relationship, the program is written using index values as Entities. This is the common denominator, the binding force of all the FORSIM subroutines.

DYNAMICS

Simulation is concerned with things flowing through a system and the dynamic interactions which result. In the example of a trucking operation, consider the relationship, at some specific time, of Empty Trucks, Available Drivers, and Shipping Orders. Let us, on the basis of the exhibited relationship, assign a Driver to a Truck and designate a Shipping Order to be transported. At the end of a certain period of time, the Truck and Driver will again become available and the Shipping Order is destroyed. This is what is meant by dynamic interaction. The dynamics are supplied by the passage of time and the occurrence of events at specific times.

In this example, we have Entities from three separate Classes. We could envision a possible future event due to each Entity; e.g., a Driver might be limited to an 8-hour drive, the Truck might be old and unreliable and its operating time might be a random variable, and the Order (cargo) might be perishable. Thus, we may wish to allow for three tentative future results of the trip, Driver fatigue, Truck breakdown, cargo spoilage. We could write short routines or Activities to deal with each contingency. To apply these routines, we would have to know the time each Driver reaches his limit, the time each Truck breaks down, etc.

Since many Drivers and Trucks are on the road at the same time, it would not be easy to assign a single future time at which the routines would act. Hence, we could assign many times, one to each Entity. Each routine then determines when it should act by examining the time slots for all Entities with which it is concerned. In other cases, it might be more convenient to associate a single time for an Activity to occur. FORSIM IV allows scope for either method. Future action times may be calculated and assigned either to each Entity or to an Activity. The main timing routine examines all the time slots and advances time to the most imminent action.

To make use of time slots, the user must define and dimension as many tables and variables as he needs. Furthermore, all time slots must be organized consecutively in core and must be addressable by a single variable name. Suppose one desires 100 slots for each of the two tables ITM1 and ITM2 and one variable JTM. One could define these and Equivalence them to a single table, ITAB. Thus, one could have ITAB (1) = JTM, ITAB (2) = ITM1, ITAB (102) = ITM2. All time slots must be integer valued. Within these limitations, one may exercise considerably freedom in naming and assigning time slots.

Time slots should be set relative to Main Clock time. They are never altered by the timing routine but merely scanned to detect the next most imminent Action Time. The Main Clock is then advanced to the selected time, and a new cycle is begun through the Activities List. Main Clock values are always available to the programmer who must reserve the first two slots of Common as integer variables. The first slot will contain the current value of Main Clock; the second slot is an indicator for recycling.

The recycle feature allows for an additional pass, or cycle, through the Activities List without advancing time. Thus, many Activities can be made to occur simultaneously, even though they do not appear consecutively in the Activities List.

A more detailed description of the timing subroutines is provided in Section III.

PROGRAM ORGANIZATION

As in all programs, one must have an initialization section, a termination section, and a "Middle" or "Working" section. In FORSIM, the "Middle" section is called the Event and Activities List.

This name is given to suggest that the user organize the time dependent or dynamic elements of his system into a sequence of routines, each dealing with some particular type of action. Each such routine is called an Activity and the sequence is called a List. The last Activity on the List should finish by calling the Main Timing routine (TIME) which will either terminate the run or cause a new cycle through the Activities List.

As previously mentioned, the time slots may be associated either with Entities or with Activities. The timing routine does not select and execute the next Activity; instead, it selects a new time for the Main Clock and causes a cycle through all the Activities. Thus, it is quite advantageous to have some sort of dynamic test at the beginning of each routine. If the test is failed, the Activity is skipped. It is also efficient to have as few time slots as possible. The model program subsequently described illustrates both these points.

SERVICES

A variety of statistical services are provided. A histogram package is available which can handle any number of separate histograms. Mean, variance, and standard deviation are calculated for each histogram. Each frequency slot contains the actual number of observations, the percentage, cumulative percentage, and cumulative remainder. A statistical analysis routine allows the accumulation and calculation of mean, variance, standard deviation, and maximum for any number of different data groups. Sampling is available from the general normal, rectangular, and exponential distributions. In addition, the user may specify any distribution provided that it can be formed in core as a cumulative probability array which can be automatically sampled as a discrete or as a continuous curve.*

To facilitate reinitialization and rerunning, routines are available which can clear or zero a set and clear out a histogram, while either retaining or altering its structure and which can reset time to zero. How many of these routines or in what combinations they are used is left to the programmer.

OTHER FUNCTIONS

The Input/Output, Arithmetic, Decision Logic, and Normal Processing functions are provided by FORTRAN IV. In many cases, a very straightforward and powerful decision logic can be concocted by interleaving FORTRAN and FORSIM commands. To pick at random from SETA an Entity whose time slot equals the Main Clock and whose first parameter is greater than 10, we gather all Entities which satisfy the conditions, into a "work" or temporary Set (SETT), and then pick an Entity at random. The following coding illustrates this.

```
      CALL ZERO (SETT)
      I = Ø
    1 CALL FIRST (I, SETA, LG)
      IF (LG) GO TO 2
      IF ((ITIM(I) . NE. MCLOK) . OR. (PAR(I, 1) . LE. 1Ø)) GO TO 1
      CALL INTO (I, SETT, LG)
      GO TO 1
```

*A set of routines provides random integers and random fractions. These STRAP-coded entities were borrowed from the STAPP System. Another gift from STAPP is the debug facility or "KEEP" subroutines. Refer to MITRE TM-67, "The STAPP System," March, 1963.

```
        2 CALL ANY (I, SETT, LG)
          IF (LG) GO TO 1Ø
```

Processing continues with the randomly selected Entity, I.

In this example, we clear out temporary Set SETT and use FIRST routine to cycle through all Entities in SETA. Acceptable Entities are put in SETT but are not removed from SETA. When SETA is exhausted, we go to 2 and pick an Entity at random from SETT. If SETT is empty, we exit to 10, since there is no Entity which satisfies the conditions. Otherwise, we continue processing with the randomly chosen Entity, I. SETT is a temporary Set in the sense that it may be zeroed and used by several different routines for temporary needs, but it still must be properly defined and dimensioned.

SECTION III
COMMAND SET

This section is a dictionary of the various command routines. A convenient tabulation of the commands may be found in Appendix B. The commands have been grouped into the following major classifications: 1) Set-Entity routines, which are the heart of the language and are discussed first; 2) Time routines; 3) Histogram routines; 4) Statistical routines; and 5) Random Number and Debug (STRAP/FAP coded) routines. The standard FORTRAN convention applies to the designation of fixed and floating-point variable names. Sets are generally denoted by ISA, ISB, or ISET, and Entities are denoted by I.

In discussing Set-Entity routines, a distinction has been made between routines which perform unconditionally and those which perform only if certain conditions are satisfied. The former are called Action routines and the latter are called Dual routines.

SET-ENTITY ROUTINES

Action Routines

COUNT (ISET, N)	Count routine causes N to be set equal to the number of entities in Set ISET.
GAINS (ISA, ISB)	Gains routine causes all entities in Set ISB, which are not already members of Set ISA, to be entered into Set ISA.
INIT (ISET)	INIT routine has the effect of initializing or emptying ISET. It must be used to initialize all Sets.

LOAD (ISET, IL, IH)	Load routine causes Set ISET to be emptied and then loads it with all Entities from IL (low bound) to IH (upper bound), inclusive.
LOSE (ISA, ISB)	Lose routine removes from ISB all Entities which are in both ISA and ISB. (ISA is unaffected.)
ZERO (ISET)	ISET is zeroed or emptied. (N.B. - Identical with INIT.)
CONVRS (ISA, ISB, IL)	CONVRS routine effectively divides a class into two mutually exclusive Sets. All Entities, from the first up to the ILth, which are not already members of Set ISB, become members of Set ISA.

<u>Dual Routines</u>

FROM (I, ISET, LG)	Entity I is removed from Set ISET. If I is not a member of ISET, then LG, logical variable, is set True. Normally LG is set False. This convention holds for all dual routines; if the action is legal, it is undertaken and LG is set False; if the action is improper, LG is set True and the command is ignored.
HEAD (I, ISET, LG)	Entity I is placed at the head of Set ISET. In a queue sense, HEAD means

	the first in line. If I is already in ISET, LG is set true.
INTO (I, ISET, LG)	Entity I is placed at the end or tail of Set ISET. In a queue sense, INTO or TAIL enters I as the last in the Set. LG is set True if I is already in ISET.
TAIL (I, ISET, LG)	Has the same effect as INTO.

<u>Test Routines</u>

EMPTY (ISET, LG)	If Set ISET is empty, then LG is set True; otherwise LG is set False. This convention holds for all test routines; if the test is satisfied, logical variable LG is set True.
EQUALS (ISA, ISB, LG)	LG is set True if Set ISA equals Set ISB, i.e., if they both have exactly the same Entities as members (although the Entities may be ordered differently in the two Sets).
IN (I, ISET, LG)	LG is set True if Entity I is contained in Set ISET.
NOTIN (I, ISET, LG)	LG is set True if Entity I is not a member of Set ISET.
WITHIN (ISA, ISB, LG)	LG is set true if Set ISA is contained within Set ISB. ISA is within ISB only if all members of ISA are also members of ISB.

Special Action Routines

SUM (ISA, IPAR, I1, I2, IRP) SUM routine forms a sum of a specific Attribute or Parameter of all the Entities in Set ISA. IPAR is the Attribute or Parameter table; it is dimensioned as an (I1, I2) array, where I1 equals the number of Entities in the given Class and I2 equals the number of Parameters to be associated with each Entity. Initially, IRP is set as the Attribute or Parameter desired; upon return, SUM sets IRP equal to the requested sum.

Example:

Let ISA contain Entities 1, 2, and 5. Let IPAR be of size (100, 5); i.e., there are 100 Entities, 5 Parameters each. We wish to sum Parameter 3 for all members of Set ISA, (ISA might contain all trucks which are on the road; Parameter 3 might contain the cargo weight carried by each truck; thus the desired sum would represent the amount of cargo in transit at any given time).

```
1. ISUM = 3
2. CALL ISUM(ISA, IPAR, 100, 5, ISUM)
```

The above code results in ISUM containing the desired sum; i.e., ISUM = IPAR (1, 3) + IPAR (2, 3) + IPAR (5, 3).

RANK (ISA, IPAR, I1, I2, IA) RANK routine reorders the Entities in Set ISA, according to Parameter IA in the associated Parameter table, IPAR (I1, I2). As in SUM, I1 equals the number of Entities, and I2 equals the number of Parameters.

Example:

Assume ISA contains the following Entity ordering (1, 2, 3) and the associated Parameter values were (5, 10, 2). Let the following queue terminology hold: TAIL (a ... z) HEAD; i.e., Entity a is last, Entity z is first. RANK will rearrange ISA into (3, 1, 2). Thus the Entity with the largest Parameter value will become first in the Set, etc.

Special Dual Routines

The special dual routines are used to pick an Entity from a Set. In a queue sense, we may choose the FIRST, LAST, or, if we want a random choice, ANY. These three routines allow one to consider a set as a FIFO, LIFO, or Random queue. The routines are Dual because, if the set is empty or exhausted, a logical variable is set True.

It is frequently desirable to find the first Entity in a Set which satisfies some set of conditions. This can be done using FIRST (or LAST, if the reverse ordering is desired), by repeatedly cycling through FIRST and by applying the set of conditions until either a satisfactory Entity is found or else the set is exhausted. This is illustrated by the example following the LAST routine described below.

ANY (I, ISET, LG)	LG is set True only if ISET is empty. If ISET is not empty, I is set randomly to some Entity in ISET.

FIRST (I, ISET, LG)	LG is set True, if ISET is empty or if a cyclic search has exhausted the Set. If I is set to zero, then the I will be set to the first Entity in ISET. If I is set to some other value upon entrance, the subroutine will search for an Entity equal to I, and then set I to the next Entity in line. Thus, repeated entrances to FIRST (if one starts by setting I = 0, and then does not alter the returned values of I) will result in an orderly search through the Set, beginning with the first and ending with the last. Termination of such a search is indicated by a True setting of logical variable LG. Care should be taken to guarantee that legitimate nonzero values of I are used during intermediate stages of the search. If an I is used which is not a member of the Set, then the search will terminate as if the Set were exhausted.
LAST (I, ISET, LG)	LG is set True if ISET is empty or if a cyclic search has exhausted ISET. If I is set to zero upon entering LAST, then I will be set to the Last Entity in the Set. If I is set nonzero upon

entering LAST, then I will be set to the next from "last" Entity after Entity I. This uses the same cyclic search technique as FIRST.

Example:

Let SETA be: LAST (5, 7, 3, 9) FIRST. Consider the following code:

```
I = 0
1 CALL FIRST (I, SETA, LG)
IF (LG) GO TO 1Ø
```

(Set of conditions, expressions, etc., in I; if I satisfies them, go to 15, or fall through and:)

```
GO TO 1
15 Routine for the First acceptable I.
1Ø Routine if no acceptable I is found
```

The above code will search and produce the first acceptable Entity or will indicate that there is none. On the first entrance, I is set to 0, becomes 9; on the second entrance, (if we loop to 1), I enters as 9, and exists as 3. This continues until an acceptable I is found or until we enter with I set to 5. If I is set to 5, upon entrance, LG is set True and the search is terminated.

Entity Moving

It is often useful to move an Entity from one Set to another. This can be done by using FROM and INTO routines described above. However, special routines are available to transfer the last or first Entity of Set A into the last or first position of Set B. This facilitates the transference of Entities between

LIFO and FIFO sets in any combination. These routines may be used conveniently when the user is interested not in the specific Entity which is moved but only in its position in the Sets.

The MOVE routines are dual purpose in that a check is made to assure that the move can be made. If it cannot, no transfer is made and logical variable LG is set True. If the transfer is legal, LG is set False and the appropriate Entity is deleted from Set ISA and inserted in Set ISB.

MOVEFF (ISA, ISB, LG)	The first Entity in ISA becomes first in ISB. If ISA is empty, no transfer is made and LG is set True.
MOVEFL (ISA, ISB, LG)	The first Entity in ISA becomes the last in ISB. If ISA is empty, no transfer is made and LG is set True.
MOVELF (ISA, ISB, LG)	The last Entity in ISA becomes the first in ISB. If ISA is empty, no transfer is made and LG is set True.
MOVELL (ISA, ISB, LG)	The last Entity in ISA becomes the last Entity in ISB. If ISA is empty, no transfer is made and LG is set True.

TIMING ROUTINES

Time slots may be associated with Entities or Sets, or they may simply be variables appearing anywhere in the simulation. How they are assigned is left to the user. However, all time slots must be consecutively ordered somewhere in memory. Furthermore, the first two slots in Common must be reserved as integer-valued fields. The first Common slot is the Main Clock and can be referenced by the programmer. The second slot is used as a recycle indicator.

RECYC

RECYC routine has no arguments. When executed, it causes the main timing routine to perform another cycle through the program without advancing time. Suppose there are two activities or routines, A and B, in the simulation and suppose that the program performs A and then B. It may be that some action in A is blocked by a condition in B, and that the blocking condition is removed when B is executed. The use of RECYC permits a second pass through both A and B, thus allowing the simultaneous occurrence of many actions before time is advanced. The following should be carefully noted:

1. Regardless of how often a RECYC is executed during a single pass, only one additional pass or cycle is made.
2. If, on a recycle pass, RECYC is again executed, a further pass will be made without advancing time.

	Hence, care must be taken in the logical positioning of the RECYC statement to prevent creation of a perpetual loop. 3. Before a recycle pass is begun, the current pass is completed.
SETIM (ITAB, ID)	SETIM routine must be used for initialization and reinitialization. It sets all time slots and the Main Clock to zero, and turns off the Recycle indicator. ITAB represents the first slot of the time-slots array. ID is the number of time slots. All time slots must be grouped consecutively as a one-dimensional table, ITAB, of dimension ID.
TIME (ITAB, ID, IRUN, LG)	TIME is the main timing routine; it should be called at the end of a program cycle. ITAB and ID, as above, represent the time-slots array and its size. IRUN is the maximum running time of the program - in whatever basic units are being used. Logical variable LG is set True only at the end of the run. TIME determines whether a recycle is to occur or time is to be advanced. In the latter case, Main Clock

is advanced to the next most imminent action time as revealed by the time-slots array. The time slots are not altered. If time has been advanced to a value less than or equal to IRUN, a new cycle is begun. If time has been advanced beyond IRUN, the run is ended. If time is less than IRUN and cannot be advanced because no future events are scheduled in the time slots array, a diagnostic statement is printed and the run is ended. A cycle is begun by returning with LG set False; a run termination (normal or error) is signalled by returning with LG set True. Thus the following code pattern should appear at the end of the simulation activities:

```
CALL TIME (ITAB, ID, IRUN, LG)
IF (LG) GO TO End-Of-Run Routine
GO TO First Activity to start a new cycle.
```

It should be stressed that the time slots are set by the programmer and are not altered by TIME; only the Main Clock is advanced.

HISTOGRAM ROUTINES

Each histogram must be defined in a Dimension statement. Dimension must be set to 5 plus the number of frequency slots desired. Next, one may call the HSTIN routine to initialize, the HSTADD routine to insert a value, and the HSTOUT routine to provide output. If desired, CLEAR routine can be called to reinitialize the histogram.

HSTIN (IHA, IN, IL, IQ)	IHA refers to an appropriately dimensioned array; IN refers to the number of slots desired; IL refers to the low value (i. e., all values $\leq$ IL will be tabulated in the first frequency slot); IQ refers to slot quantization (i. e., the second slot will tabulate values $>$ IL and $\leq$ IL + IQ, etc.; the last slot will tabulate values $>$ IL + (IN-2) IQ). Note that all arguments are fixed-point integer valued. HSTIN is used to initialize and set up the structure of a histogram and must be executed before calling HSTADD for a specific histogram. It will be recalled that a histogram array must be dimensioned five slots greater than the number of frequency slots desired. The first three slots are used to store IN, IL, and IQ. Slots 4 and 5 are used by HSTADD to accumulate ΣN and ΣN^2

	where N represents the values tabulated. This allows the mean and the standard deviation to be calculated.
HSTADD (IHA, IVL)	IHA is the variable name assigned to a histogram; IVL is the value (fixed point) to be tabulated. HSTADD enters IVL and IVL^2 into running summations to form the mean variance and the standard deviations. It then determines the appropriate frequency slot for IVL and increments it by one.
HSTOUT (IHA, TTL)	HSTOUT routine causes histogram IHA to be printed out under the heading given in TTL. TTL may be six alpha-numeric characters. Histogram printouts tell the number of entries, the mean, the standard deviation, the variance, and, for each frequency slot, the number of observations, the relative frequency, the cumulative probability, and the cumulative remainder.
CLEAR (IHA)	CLEAR routine zeros all frequency slots and running totals for N and N^2 but does not affect the values of IN, IL, and IQ. Thus a histogram can be

cleared for future use involving the same frequency slot structure. If a new structure is desired, HSTIN must be used in addition to CLEAR.

STATISTICAL ROUTINES

Distributions

EXPN (IVL, FMN)	EXPN routine returns with IVL (fixed point) set to a randomly chosen value from an Exponential distribution of mean value FMN (floating point). Since EXPN is expected to be used for inter-arrival times, IVL will never be set less than 1.
NORMAL (IVL, FMN, SD)	NORMAL routine returns with IVL (fixed point) set to a randomly chosen value from a Normal distribution of mean value FMN and standard deviation SD (both floating point).
RECT (IVL, MN, MX)	RECT routine returns with IVL set to a randomly chosen integer in the range $MN \le IVL \le MX$. All arguments are integers.

In addition to the above, continuous or discrete sampling may be had from any array read or formed in core. The array must represent a cumulative probability distribution and must be dimensioned as (2, N), where N is the number of values given. Data cards containing the distribution can be read in through standard matrix array format. Thus the data cards should contain the following ordering:

Probability 1, Value 1, Probability 2, Value 2 .... Probability N, Value N. In core, ARRAY (1, J) = Probability J; and ARRAY (2, J) = Value J.

FORTRAN permits arrays to be easily read, consequently, no special provision has been made for input. The arrays should be given floating-point variable names and properly dimensioned. Probability 1 must always equal 0.0; Probability N, where there are N points in the curve, must always equal 1.0.

CDST (IVL, TAB, 2, N)	CDST routine provides a fixed-point value, IVL, chosen randomly from the array, TAB, of dimension (2, N). Sampling is continuous in the sense that linear interpolation is performed between adjacent points on the curve.
DDST (IVL, TAB, 2, N)	DDST routine is similar to CDST in arrangement of arguments. It performs discrete sampling. A random number, X, is chosen in the range $0.0 \leq X \leq 1.0$. If Probability $i < X \leq$ Probability $i+1$, then Value $i+1$ is chosen, converted to fixed point, and returned as IVL.

Fixed-point values are returned by all of the previous statistical routines. If floating-point values should be needed, the subroutine in question can be reassembled with an INTEGER IVL card inserted.

Data Analysis

Data is frequently accumulated for which Mean, Variance, Maximum, and Standard Deviation data are desired. The following two routines provide an easy and rather general method for obtaining this data.

STATS (SARAY, XP, IDX)	STATS routine accumulates ΣXP, ΣXP^2, ΣN, and M, where XP is floating-point data, ΣN is the number of XPs entered, and M is the maximum XP encountered. To accomplish this, STATS requires the following: SARAY, a floating-point array of dimension 4, which must be defined by the user; XP, a floating-point variable or linear variable array containing the accumulated data; and IDX, the dimension of XP, i.e., the number of datums stored in XP. If XP is a single variable, IDX = 1. If XP is an array, all values of XP(I), I = 1, IDX will be tabulated. Thus one may dimension XP equal to 100, fill only the first 25 slots, and call STATS with IDX = 25; or one may fill slots 76 to 100, and call STATS with IDX = 25 and XP set as XP(76). The table, SARAY, supplied by the user, is utilized to store running summations of the number of data,

their values and their values squared, and the maximum value encountered.

STOUT (SARAY, TTL)	STOUT routine prints the number of entries accumulated, the mean value, the variance, the standard deviation, and the maximum value, using the data stored in SARAY. TTL may be six alphanumeric characters and is used as a heading for the statistical printout. SARAY values are not affected by STOUT. One can print intermediate results and continue the accumulation. If SARAY is to be used for a new accumulation, the four slots of SARAY must be zeroed--a two statement DO loop will suffice.

STRAP/FAP-CODED ROUTINES

The debug printing facility and the random number generators are taken intact from the STAPP system.* Except as noted they are available in both STRAP and FAP.

*Fagan, G. A., "The STAPP System," MITRE TM-67, March 1963.

KEEP - Debug Printing Routine

Purpose

The KEEP routine prints, under certain conditions, an identification code of eight or less hollerith characters, followed by either a number of specified single words or a block of specified size, starting at a named origin. Numerical values are printed seven or less words per line, seventeen print wheels per word, with seven decimal places and in fixed-point format.

Calling Sequence

(a) CALL KEEP (6HXXXXXX, KEYWD, N, ITEM1, ITEM2, ITEM3,--)
Provided that bit number N ($12 \leq N \leq 49$)* of location KEYWD is 1, the line

```
XXXXXXXX     A     B     C
```

will be printed, where A is the value of ITEM1, etc.

(b) CALL KEEP (4HYYYY, KEYWD2, - M, ITEM, K)
Provided that bit M of KEYWD2 is 1, the identification YYYY will be printed, followed by the K words, ITEM through ITEM (K). (The negative sign attached to M in the calling sequence denotes block print.) In both cases, seven numerical values per line may be printed, and multiple lines are generated if required.

* For 7090 use, the KEY word is an octal word with bit 0 for the sign bit, etc. The appropriate bit range is ($0 \leq N \leq 35$). In type "b" calling sequences, where the negative sign denotes block printing, the bit number must be numeric rather than symbolic. Identification on the 7090 is limited to six or fewer hollerith characters.

(c) CALL KEEP1 (A, B, C)

This calling sequence superimposes an additional control over that given by the bit and key word which are specified in every calling sequence. KEEP calls encountered after the execution of the KEEP1 call will print only if their appropriate control bit is 1 and if $B \leq A < C$.

This is convenient for suppressing debug printing during certain portions of a program. Usually, A is the "current time" and B and C are preset limits; however, other values may be used. A, B, and C may be symbolic or numeric, fixed or floating point.

NOTES

1.[*] The identification must consist of eight or less hollerith characters.
2.[*] The KEY word is a FORTRAN integer with bit number 60 for the sign bit, etc.
3.[*] The bit number is a fixed-point integer (12 through 49). In type "a" calling sequences, it may be numeric or symbolic.
4. The calling sequences may call for any number of words to be printed; any number greater than seven will be printed on successive lines, seven words per line.
5. The equivalent FORTRAN format is:

```
(1H0,A8, 1X,7F17.7/(10X, 7F17.7))
```

[*] For 7090 use, the KEY word is an octal word with bit 0 for the sign bit, etc. The appropriate bit range is $(0 \leq N \leq 35)$. In type "b" calling sequences, where the negative sign denotes block printing, the bit number must be numeric rather than symbolic. Identification on the 7090 is limited to six or fewer hollerith characters.

6. CALL KEEP (6HXXXXXX, Key, N) is permissible, and will print only the identification.
7. The program enters the FORTRAN routine (IOCS) to execute printing.

Additional KEEP Calls for 7030 Only

The following four KEEP types are only available for the 7030 and can be used in either formats (a) or (b) under Calling Sequence above (i. e., individual or block printing).

(a) CALL KEEPA - This causes the chosen words to be printed out in alphabetical format. No blanks or spacing are inserted between successive words on a line. Fifteen 7030 words per line are printed.

(b) CALL KEEPE - The selected values are printed in normalized floating-point format, e.g., 0.5076E22. Six values per line are printed.

(c) CALL KEEPI - The selected values are printed as integer numbers, seven words per line.

(d) CALL KEEPO - The selected values are printed as octal numbers, five words per line. This format provides a convenient method for printing logical variables.

KRED

A set of Random-Number Generators (FRED, FNOD, STRN, PURN). KRED is a multiple-entry subroutine which generates a sequence of pseudo-random numbers. The three entries, KRED, FRED, and FNOD, are called as FORTRAN functions and generate random numbers. The two additional entries, STRN and PURN, are used for initialization and termination, respectively, and are called subroutines.

Examples:

(a) CALL STRN (IRN, KEYW, BIT) - This is an initializing entry and must be called before KRED, FRED, and FNOD are used in a machine run. The initial random number, which must be previously stored in the location IRN, is moved to KRED's internal storage. The initial random number must be an octal word ending in 1 or 5 (i.e., integers of the form 4K + 1). The locations of the Key word and the control bit which govern the debug printing for KRED, etc., are then stored in appropriate locations.

(b)* I = KRED (ZZ) - This FORTRAN statement causes a random integer to be generated and stored in location I. Provided that the appropriate bit, specified in the STRN call, is 1, the subroutine will print the identification RANINT at location ZZ (up to eight characters), followed by the random integer.

(c)* A = FRED (ZZ) - A random floating-point fraction (in the range of 0 to 1) is generated and stored in location A. The identification XXXXXX at location ZZ and the random fraction may be printed by KEEP.

(d)* VAL = MEAN + FNOD (ZZ) * STDEV - FNOD generates a normal deviate with a mean of zero and standard deviation of 1. The above statement, therefore, causes a single value from a distribution with a mean of MEAN and a standard deviation of STDEV to be stored in location VAL. Printing of the identification and the normal deviate are controlled by KEEP.

*When using KRED, FRED, or FNOD, some argument must appear or the FORTRAN compiler will believe it is dealing with a variable and not a function. Any dummy argument will suffice, and setting KEYW to zero will inhibit printing. One could write I = KRED (ZZ) where ZZ is defined in a DATA statement as 6HRANINT.

(e) CALL PURN (INO) - The last random number generated in the sequence by KRED, FRED, or FNOD, is moved from internal storage to the location INO. It is convenient to use this to obtain the values of "starting random number" at the beginning of each simulation run, so that an individual run can be duplicated in case of difficulty or error.

SECTION IV
SOME TECHNIQUES

GORDON TYPE FACILITIES AND INTERRUPTS

While FORSIM IV is not quite as simple to use as the Gordon Simulator, it is much more flexible. Hence, facsimiles to Gordon features can be created.

As exhibited in the model program (Section VI), a simple server, or facility, can be constructed as a Set limited to only one Entity at a time. A Set which can contain only two Entities can be treated as a facility with interrupt provision. A large variety of interrupts can be constructed, e.g., interrupts to any depth, dependent on the relative priority of the Entities; this would involve a Ranked Set and Entities whose priorities are stored as Attributes. To clarify this technique, a method of achieving a Gordon-type interrupt is described below.

Consider a facility Set, IFSET, limited to two entities. Associated with IFSET are an Interrupt Indicator, IND, and two time slots, IT1 and IT2.

(a) A non-Interrupt Entity may enter IFSET only if the Set is empty. If it enters, IT1 is set to PTM + MCLOK = Departure time, where PTM is the passage time through the facility.

(b) An Interrupt Entity may enter IFSET only if another Interrupt Entity is not currently using IFSET. If IND = 1, the facility is interrupted and cannot be disturbed. If IND = 0 and/or the Set is empty, the Entity enters, sets IT1 to departure time and sets IND to 1 to prevent an interrupt. If IND = 0 and the Set is not empty, an interrupt is effected. IND is set to 1; the Interrupt Entity is moved into the first slot of IFSET, thus automatically pushing the current Entity to a lower slot; PTM is calculated, IT2 is set to PTM + IT1, and IT1 is set to PTM + MCLOK.

(c) Exit from the facility is accomplished at time MCLOK = IT1. The first Entity in IFSET is removed and IND is set to 0. If IFSET is empty, no further action is required. If IFSET is not empty, then set IT1 equal to IT2. Thus the interrupted Entity is returned to a first position in IFSET, and its Departure time has been recalculated to account for the interrupt. Although two time slots are used, only IT1 can cause an action, i.e., a Departure. Hence only IT1 need appear in the time table.

The following example demonstrates this type of Interrupt facility. Consider a facility, ISV, which normally processes data from queue IQB. Assume a flow of priority data which can interrupt normal processing. This flow enters queue IQA but is delayed there only if denied immediate access on ISV. Passage through ISV is always 40 TIME units.

```
C                        ISV  OUTPUT ROUTINE
        50               IF (ITIME(5) . NE . MCLOK)GO TO 60
                         MOVEFL (ISV, INXT, LG)
                         IND  =  0
                         CALL EMPTY (ISV, LG)
                         IF (LG) GO TO 60
                         ITIME(5)  =  ITM

C                        ISV  INPUT  ROUTINE
        60               CALL EMPTY (ISV, LG)
                         IF (.NOT. LG) GO TO 70
                         MOVEFF (IQB, ISV, LG)
                         IF (LG) GO TO 70
                         ITIME(5)  =  40 + MCLOK
```

```
C                           PRIORITY ROUTINE
       70                   (CODING)
                            CALL INTO (I, IQA, LG)
                            IF (IND. EQ. 1) GO TO 80
                            MOVEFF (IQA, IQB, LG)
                            IND  =  1
                            ITM = ITIME(5) + 40
                            ITIME(5) = 40 + MCLOK

C                           NEXT ROUTINE
       80
```

DETERMINING PASSAGE TIMES THROUGH A SET

Maximum and average passage times through a Set can be obtained with the STATS and STOUT routines. To accomplish this, set a Parameter or Attribute equal to Main Clock time as an Entity enters the Set. When an Entity is removed from the Set, form XTM = difference between the parameter time and the present Main Clock time (NOTE: XTM should be floating point, although Main Clock is fixed point). If an appropriately named SARAY of dimension 4 has been set up initially, one can then call STATS (SARAY, XTM, 1).

When a printing is desired, say at the end of the run, call STOUT. In addition to maximum and average passage times, one also obtains the number of Entities which have passed through the Set, the variance, and the standard deviation. Passage time calculations are performed only on Entities which have passed through the Set; no account is made of those still in the Set at printout time. Of course, these can be included simply by forming XTM and calling STATS for all Entities still in the Set just prior to calling STOUT. (The model program contains an example of a Passage Time calculation.)

CLASSES OF SETS

Although we cannot formally treat Classes of Sets, we can effectively treat them by utilizing a simple FORTRAN technique, and, using Trucks and Garages as Entities, proceed as follows: A Garage is either full or empty. There are two Sets for Garages (FULL, OPEN), and all Garages are in one or the other Set. A Truck is either on the road on in one of the Garages. Assuming 10 Garages, there are 11 Sets for Trucks (ROAD, GAR1, GAR2 ... GAR10). Assume there are 30 Trucks, each of which is either on the road or in some garage. To remove Truck IT from the ROAD and place it in some randomly chosen Garage that is not full, we code as follows:

```
CALL ANY (IGAR, OPEN, LG)
CALL FROM (IT, ROAD, LG)
CALL INTO (IT, GARS (1, IGAR), LG)
```

Thus Truck IT is placed in the randomly chosen Garage IGAR. We can do this if we make use of a trick in dimensioning the Sets GAR1 ... GAR10. First we dimension GAR1 (31) ... GAR10 (31); the normal procedure. Now we dimension GARS (31, 10) and equivalence GARS (1, 1) = GAR1, GARS (1, 2) = GAR2 ... GARS (1, 10) = GAR10. This simple technique allows us to use GARS as a "Class" of the Set-Entity Garage. We may refer directly to GARS or indirectly to GARS (1, 5); in either case the result is the same. The procedure is quite convenient if one has a large number of Sets and wishes to index into them.

In initialization, we must initialize all 10 Sets only once, i.e., as GARS (1, 1) ... GARS (1, 10) or else as GAR1, GAR2 ... GAR10.

SECTION V
SYSTEM OPERATION

MAINTENANCE

From the previous explanation, the FORSIM system is shown to be a set of subroutines which are treated like commands. The subroutines, which are described in Section II, are written in FORTRAN and are quite brief. The user may obtain copies of the binary and the symbolic decks. An existing subroutine can be modified simply by examining its listing, making a few changes, and reassembling. A new subroutine can be coded in FORTRAN, assembled, and placed with the other binary decks. Only the random number generators and the debug routines are not amenable to modification in this fashion. The maintenance and expansion of the system can be done easily by the user.

Another advantage of a FORTRAN package over a precompiler language is the ease of converting it for use in other computer systems. Since the STRAP coded routines are also available in FAP, FORSIM IV is immediately available for use on any IBM 7030 or 7090 which can compile FORTRAN IV.* Since most simulation-oriented organizations have a random number generator for their computer, and since most computers have some sort of debug facility (one can always use lots of FORTRAN print statements), FORSIM IV probably could be modified and reassembled for use on any computing system capable of compiling

*There are a few incompatibilities between FORTRAN IV for the 7030 and for the 7090. In the 7030, DATA statements use parentheses, whereas slashes are used in the 7090; the statement, PRINT N, obtains the community output tape on the 7030, and the 7090 equivalent is WRITE (JT, N) where JT is normally 6. These incompatibilities occur on ten cards in FORSIM IV and have been corrected to give a 7090 symbolic deck. It has been assumed that community output is on tape number 6. If this is not the case, it is necessary to redefine the variable JT at three places (Card numbers 04165, 04855, and 06105). In any event, JT must refer to a BCD mode output tape.

FORTRAN IV. Modification would consist of inserting decks for the appropriate random number generators and debug facility (if any), modifying the code for the Statistical Distribution Subroutines, which call for random numbers via FRED and KRED, and reassembling all subroutines. The time spent in such a conversion should not be excessive.

Since the benefits of FORTRAN are well established, it seems hardly necessary to mention that subroutines, functions, chaining, etc., are all possible with FORSIM IV. Anything which can be written in FORTRAN can be written in FORSIM. One must simply be sure to include the FORSIM subroutines when they are used, otherwise he writes a FORTRAN IV program. The running of the program is just like the running of a FORTRAN program. FORSIM binary decks can be inserted with one's problem deck, or can be placed on a systems tape. They are treated like any other subroutine deck.

PROGRAM INITIALIZATION CHECKLIST

This section provides a concise summary of proper procedures for starting a FORSIM program.

(a) Before using any other Set commands, all Sets must be initialized by either CALL INIT or CALL ZERO.

(b) All histograms must be initialized by calling HSTIN. If a histogram is being reinitialized, one must first call CLEAR and, if a new structure is desired, call HSTIN.

(c) SETIM must be called before attempting to use Main Clock. This is essential since the loader places meaningless information in Common.

(d) All Sets must be dimensioned to 1 + the number of slots desired; e.g., if SETA is to hold at most 10 Entities, then dimension SETA to 11 words.

(e) All histogram arrays must be dimensioned to 5 + the number of frequency slots desired; e. g. , if HSTA is to have 30 slots, then dimension HSTA to 35 words.

(f) All statistical arrays must be dimensioned to length 4 words prior to calling STATS.

(g) All Parameter or Attribute arrays must be dimensioned as PAR (M, N) where M = number of Entities in the Class and N = number of Parameters to be associated with each Entity.

(h) All time slots must be properly dimensioned and formed as a consecutive block in core, referenced by a single fixed-point variable name.

(i) Common must be reserved for the Main Clock and Recycle Indicator.

(j) At least one logical variable name should be defined for use with Test and Dual commands. Any deviations to normal FORTRAN conventions for fixed- and floating-point names should be defined.

(k) Before using the Random Numbers or the Statistical Distribution commands, the random number stream must be initialized by CALL STRN (IRN, KEYWD, N) where IRN, the initial random number, is an integer of the form 4K + 1. Normally KEYWD should be set to zero to prohibit printing the random numbers.

SECTION VI
MODEL PROGRAM

PROGRAM DESCRIPTION

The Model Program exhibits FORSIM applied to a very simple simulation problem. Figure 1 is a Message Processing system in which two input-message lines and one output-message line are available. One type of input message is stored randomly on a regenerative drum of very large capacity. This procedure is represented by a random queue without a capacity restraint but with a time limit. These Type 1 messages are created at a poisson rate with exponential interarrival times. The Type 2 input messages are also poisson but these are stored in a sliding register bank. There is no time limit on retaining messages but there is a capacity restriction of 30 messages. This is represented as a FIFO queue.

Messages are selected from the Random queue or, if it is empty, from the FIFO queue to be processed singly, and are transmitted out of the system. We desire a statistical analysis of the number of messages of each type which are lost per second and of the passage times through the Random queue, as well as histograms of the age of each type of message sent.

The program is designed for multiple runs. The user specifies how many runs on the first data card and, using one card per run, specifies the mean of exponential input interarrival time, the normal mean output rate and the standard deviation of output rate, histogram structures, and running time (in basic units of milliseconds). The messages are set up as a Class (i. e., each Message is an Entity) of 1000 Entities. This implies that the system will never contain more than 1000 Entities at the same time. There is no limit (other than machine time budget) on the number of messages which may pass through the system during a run.

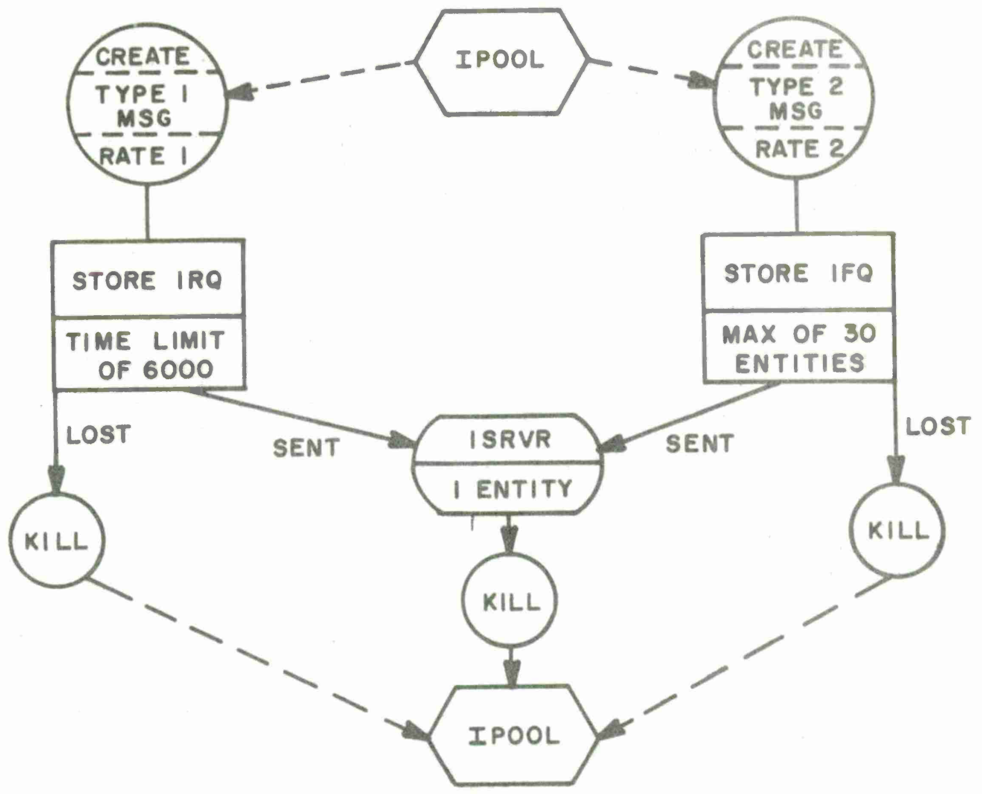

Fig. 1. Model Program

The technique of creating and destroying Entities is demonstrated by the use of the Set, IPOOL. IPOOL is not a part of the simulated system but a depository for all Entities not presently in the system. Entities are created, or inserted in the system by removing them from IPOOL; entities are destroyed, or removed from the system by inserting them in IPOOL. Thus, IPOOL serves as a revolving fund of Entities.

Class:	Messages of 1000 Entities
Sets:	IPOOL - General Depository for messages. (dimension = 1000)
	IRQ - Random Queue (dimension = 1000)

	IFQ	- FIFO Queue (dimension = 1000)
	ISRVR	- Single Server Facility (dimension = 1)
Parameters:	PAR -1-	Time in system per Entity
	-2-	Type 1 (random) or Type 2 (FIFO) Message.
Statistics:	SRLST	- Random Entities lost per second.
Arrays:	SFLST	- FIFO Entities lost per second.
	SRTM	- Passage Time for Random Entities sent.
Histograms:	IHRQ	- Time in system for Random Entities sent.
	IHFQ	- Time in system for FIFO Entities sent.
Time	ITIME (1)	- Arrival time for next Random Message.
Slots:	(2)	- Arrival time for next FIFO Message
	(3)	- Output time for the Message in the Server.
	(4)	- Time to perform per second statistics.

Times must be associated with each Entity for statistical purposes. While this could be done by creating time slots for each Entity, the procedure would be quite inefficient since the Main Timing routine would have to scan a Time Table of over 1000 slots. Hence, time is made a Parameter (Entity Attribute) and the Time Table is kept to four slots. It should also be noted that each Activity begins with some dynamic test. This enables the program to cycle efficiently through the List, bypassing Activities not due for action.

PROGRAM EXPLANATION

A summary of program flow organization is presented in Table 1. The program listing is shown in Appendix I.

Definitions and Dimensions

In this section of the program, we define and dimension all Sets, Parameter arrays, and Statistical and Histogram Arrays. Common space is reserved for

Table 1

ORGANIZATION OF PROGRAM FLOW

1. Definitions and Dimensions
2. Run Initialization (begins at Statement 1)
3. Activities List
    a. Random Queue Input
    b. FIFO Queue Input
    c. Random Queue Overflow
    d. FIFO Queue Overflow
    e. Server Input
    f. Server Output
    g. Statistics Compilation
    h. Cycle Termination (exits to 3a or 4)
4. End-of-Run Routine
    a. Run Output Printing (exits to 4b or 4c)
    b. Rerun Preparation (exits to 2)
    c. Job Termination

the Main Clock and the Recycle indicator. Variable names are defined if they differ from the usual fixed/floating-point convention. All Sets and Time are initialized. The Random Number Generators are initialized by STRN, and the number of runs is determined. This section defines all of the initial conditions of the simulation.*

*Note that STRN is called only once, regardless of how many streams of random numbers are generated, and that KEYWD is set off to inhibit printout of random numbers. Also all Sets are dimensioned to 1 plus the number of Entities, while the Parameter Array has exactly the number of Entities as its first dimension.

Run Initialization (Statement 1 of the Program Listing)

This section reads all data for a run, sets up the histogram structures, and loads IPOOL with all Entities. Time Slot 4 is set to accumulate statistics at the end of the first second (i. e., at 1000 milliseconds). NRAN is used to count the number of runs performed. This section is executed only once per run.

Begin Activities List

Random Queue Input (Statement 10)

ITIME(1), the first time slot, equals MCLOK when a message is due to be created. If this Activity is executed, an Exponential interarrival time, IT, is determined and the time slot is reset. An Entity is moved from IPOOL to IRQ. Since we must set the Entity parameters, we must know which specific Entity was moved. The new Entity has been placed in the first slot of IRQ, hence calling FIRST sets I to the Entity. Since we are confident that there will always be an Entity in IPOOL, and that we will never double insert an Entity in IRQ, we ignore the settings of logical variable LG.

FIFO Queue Input (Statement 20)

This section is logically similar to the preceding Activity. Since we desire a FIFO Queue, we place the new Message, Entity, in the last place of IFQ. In both Input Activities, a message is generated immediately at time zero. If an initial lag time were desired, the appropriate time slots could have been set in the Run Initialization section. An alternative input scheme would employ a tape whose records contain a time and a type code. If the tapes were ordered by time, a single Activity would read it and enter a message of the proper type at the indicated times.

Random Queue Overflow (Statement 30)

The Random Queue can overflow only if a message remains in it longer than six seconds. Time in the queue is calculated for each message in the Set, IRQ, until LG signals that the Set is exhausted. Control then passes to the next Activity. Overtime messages are removed and returned to IPOOL. STR counts the number of messages lost. Since we desire a per/second statistical breakdown, STR is accumulated here and analyzed in a later Activity.

FIFO Queue Overflow (Statement 40)

If more than 30 messages are in IFQ, then at the time overflow occurs, the 31st message must have just been inserted. The oldest (i.e., the first insertion) is removed and returned to IPOOL. STF count is augmented for the per/second breakdown.

Server Input (Statement 50)

If the server (Set, ISRVR) is not empty, we pass to the next Activity. If it is empty, we first attempt to fill it by a randomly chosen message from IRQ. If IRQ is empty (i.e., LG is Set True), we attempt to draw from IFQ. If both queues are empty we pass on. If we supply from IFQ, we move the first message in IFQ (since it is FIFO queue) to ISRVR and calculate its processing time as a normal deviate. The third time slot is set to signal the end of processing on the server. If we supply from IRQ, IMSG is set to a randomly chosen message. We remove it, place it in the server, and calculate its processing time. We also determine its passage time, PTM, or the time it spent in IRQ. This value is then entered for statistical analysis (STATS).

Server Output (Statement 60)

If the time slot is not equal to MCLOK, then either there is nothing in the server or a message is still being processed. In either case, we exit to the next Activity. We also exit if the server is empty. This double check is necessitated by the recycle feature. If there is a message due to be sent out, we remove it (only one message can be in ISRVR, hence calling ANY routine retrieves it) and return it to IPOOL. We then call RECYC. At the end of this cycle, a recycle will be made. If there were data in either queue, it would not have been able to enter the server, since it was full. The second cycle will catch any such message before advancing time. However, if both queues are empty and time is unchanged, we would pass the first check of the Server Output Activity and then try to empty the server a second time. RECYC would be called again, another cycle made, and a perpetual loop would result. Thus we employ a double check for entrance to this Activity. When this Activity is executed, we collect time-in-system data for the appropriate histograms, and then exit.

Statistics Compilation (Statement 70)

The fourth time slot is used to determine one-second intervals. At the end of each second, we enter STATS for the per-second statistical analysis of lost data. INSZ is another double check to prevent multiple entries on recycles. The lost data counters are reset to zero to begin accumulation for the next second.

Cycle Termination (Statement 80)

This is the last Activity of the Activities List. It must be entered every cycle and recycle. TIME will advance Main Clock (MCLOK) or not, as indicated by the absence or presence of a Recycle. If the run is over, LG is set True and we either exit to the End-of-Run Routine or we start another cycle by looping to the first Activity (by using Statement 10).

End-of-Run Routine (Statement 90)

End-of-Run printing is done under a heading telling what run has just been executed. Histograms and Statistical analysis are printed using appropriate six-character headings. If all the runs have been executed, we go to statement 99 and terminate the job. Otherwise, we zero the three Sets in the system, clear out the Statistical and Histogram arrays, and reset time to zero. Control then passes to the Run Initialization routine.

It should be noted that if the last value of MCLOK (in milliseconds) is not a multiple of 1000 (i.e., the run does not end exactly on a second), the lost-messages data accumulated over the last second is lost. A routine could be written to include this data if desired. This point should be carefully noted. If time slots are set at 10,001, 9,999 and IRUN is set at 10,000, the data will be lost. The last cycle is at time 9,999 and the Statistics Compilation Activity is not entered. The TIME routine cannot advance time within the bounds of the Run time limit, so it signals End of Run. Of course, the last cycle might have been at 9,005. A routine to include the marginal lost data must contend with both contingencies.

APPENDIX I

```
C                 FORSIM IV  MODEL  PROGRAM
C
C
C             DEFINITIONS  AND  DIMENSIONS
      DIMENSION IPOOL(1001),IRQ(1001),IFQ(1001),ISRVR(2)
      DIMENSION SRTM(4),SRLST(4),SFLST(4),IHRQ(35),IHFQ(35)
      DIMENSION ITIME(4),PAR(1000,2)
      COMMON MCLOK,MCYC
      LOGICAL LG
      INTEGER PAR
      KEYWD = 0
      CALL STRN(5,KEYWD,48)
      CALL ZERO(IPOOL)
      CALL ZERO(IRQ)
      CALL ZERO(IFQ)
      CALL ZERO(ISRVR)
      CALL SETIM(ITIME,4)
      READ 100, NRUNS
C
C             SET UP POOL INITIALLY AND READ RUN PARAMETER,ETC..
    1 READ 101, IRUN,FMN1,FMN2,FMN3,SD,IL1,IQ1,IL2,IQ2
  100 FORMAT(6X,I3)
  101 FORMAT(6X,I5,4(1X,F6.2),4(1X,I3))
      ITIME(4) = 1000
      CALL HSTIN(IHRQ,30,IL1,IQ1)
      CALL HSTIN(IHFQ,30,IL2,IQ2)
      NRAN = NRAN + 1
      CALL LOAD(IPOOL,1,1000)
C
C             SIMULATION  CYCLE  BEGINS
C
C          INPUT  TO  RANDOM  QUEUE
   10 IF(MCLOK.NE.ITIME(1))GO TO 20
      CALL EXPN(IT,FMN1)
      ITIME(1) = MCLOK + IT
      CALL MOVEFF(IPOOL,IRQ,LG)
      CALL KEEP(6HINPRAN,KEYWD,48)
      I = 0
      CALL FIRST(I,IRQ,LG)
      PAR(I,1)=MCLOK
      PAR(I,2) = 1
C
C          INPUT TO  FIFO  QUEUE
   20 IF(MCLOK.NE.ITIME(2))GO TO 30
      CALL EXPN(IT,FMN2)
      ITIME(2) = MCLOK + IT
      CALL MOVEFL(IPOOL,IFQ,LG)
      CALL KEEP(6HINPFIF,KEYWD,48)
```

```
C             OVERFLOW   RANDOM   QUEUE
   30 IMSG = 0
   31 CALL FIRST(IMSG,IRQ,LG)
      IF(LG)GO TO 40
      IF((MCLOK - PAR(IMSG,1)).LT.6000)GO TO 31
      STR = STR + 1.0
      CALL FROM(IMSG,IRQ,LG)
      CALL INTO(IMSG,IPOOL,LG)
C
C             OVERFLOW   FIFO   QUEUE
   40 CALL COUNT(IFQ,KT)
      IF(KT.LE.30)GO TO 50
      CALL MOVEFF(IFQ,IPOOL,LG)
      STF = STF + 1.0
C
C             SERVER   INPUT
   50 CALL EMPTY(ISRVR,LG)
      IF(LG)GO TO 51
      GO TO 60
   51 CALL ANY(IMSG,IRQ,LG)
      IF(LG)GO TO 55
C         SUPPLY   FROM   RQ
      CALL KEEP(6HSUPRAN,KEYWD,48)
      CALL FROM(IMSG,IRQ,LG)
      CALL INTO(IMSG,ISRVR,LG)
      CALL NORMAL(MTM,FMN3,SD)
      ITIME(3) = MTM + MCLOK
      PTM = MCLOK - PAR(IMSG,1)
      CALL STATS(SRTM,PTM,1)
      GO TO 60
   55 CALL EMPTY(IFQ,LG)
      IF(LG)GO TO 60
C         SUPPLY   FROM   FQ
      CALL KEEP(6HSUPFIF,KEYWD,48)
      CALL MOVEFF(IFQ,ISRVR,LG)
      CALL NORMAL(MTM,FMN3,SD)
      ITIME(3) = MTM + MCLOK
C
C             SERVER   OUTPUT
   60 CALL KEEP(6HOUTPUT,KEYWD,49,ISRVR(1),IRQ(1),IFQ(1)
     1PTM,NTM,MCLOK)
      IF(ITIME(3).NE.MCLOK)GO TO 70
      CALL EMPTY(ISRVR,LG)
      IF(LG)GO TO 70
      CALL ANY(IMSG,ISRVR,LG)
      CALL MOVEFF(ISRVR,IPOOL,LG)
      CALL RECYC
      NTM = MCLOK - PAR(IMSG,1)
      IF(PAR(IMSG,2).EQ. 2 )GO TO 64
      CALL HSTADD(IHRQ,NTM)
      GO TO 70
   64 CALL HSTADD(IHFQ,NTM)
C
C             COMPILE STATISTICS PER SECOND
   70 IF(ITIME(4).NE.MCLOK)GO TO 80
      IF(INSZ.EQ.MCLOK)GO TO 80
      INSZ = MCLOK
      ITIME(4) = MCLOK + 1000
      CALL STATS(SRLST,STR,1)
      CALL STATS(SFLST,STF,1)
      STR = 0.0
      STF = 0.0
C
C              END A CYCLE
   80 CALL TIME(ITIME,4,IRUN,LG)
      CALL KEEP(6H TIMET,KEYWD,-48,ITIME,4)
      IF(LG)GO TO 90
      GO TO 10
C
C             END OF RUN RTN.
   90 PRINT 102, NRAN
  102 FORMAT(10X/10X/40X,15HOUTPUT FOR RUN ,I3)
      CALL HSTOUT(IHRQ,6HRMSENT)
      CALL HSTOUT(IHFQ,6HFMSENT)
      CALL STOUT(SRLST,6HRMLOST)
      CALL STOUT(SFLST,6HFMLOST)
      CALL STOUT(SRTM,6HRQTIME)
      IF(NRAN.EQ.NRUNS)GO TO 99
C
C             SETUP FOR NEW RUN
      CALL ZERO(IRQ)
      CALL ZERO(IFQ)
      CALL ZERO(ISRVR)
      DO 92 I = 1,4
      SRTM(I) = 0.0
      SRLST(I)= 0.0
   92 SFLST(I)= 0.0
      CALL CLEAR(IHRQ)
      CALL CLEAR(IHFQ)
      CALL SETIM(ITIME,4)
      GO TO 1
C
C             END OF JOB
   99 RETURN
      END
```

APPENDIX II

SIMULATIONS - LARGE AND SMALL

FORSIM IV is designed for small to medium size simulations. The goal is clarity and simplicity. After all, a simulation language is a way of looking at reality, a guide to organizing or directing one' s systems thinking toward the construction of a model. To ease the analysis of a problem, the language should provide a conceptual background that is simple yet flexible and powerful. To ease the coding, there should be several, even redundant, commands to permit individual expression. Each command should be simple in itself; the sum total should be a rich fund from which one picks to suit his needs. FORSIM IV is an exemplification of this philosophy.

While FORSIM is most convenient in obtaining quick results for moderate problems, it can be expanded to handle quite large problems. Large simulations are not always a matter of bit-packing. They usually entail considerable logical complexity, possibly requiring that several programmers work on various sections of the problem. In such cases FORSIM can be advantageously combined with FAST.* FAST is a precompiler to FORTRAN and allows one to define all bit-packing and table dimensioning once in a central dictionary. Individual programs are precompiled against the dictionary to achieve common linkages. Structurally, changes to a system can often be effected with the dictionary rather than the programs, thus shifting much routine maintenance from the programmers to a technical assistant.

*Refer to MITRE SR-24, "FAST, FORTRAN Automatic Symbol Translator," January, 1962.

This report does not contend that FAST/FORSIM is a panacea for large simulations. It is not; nor is anything else. It is a feasible approach for some problems. FORSIM is very close to FORTRAN and is based upon simple set-theoretic concepts. It provides a conceptually simple and flexible base. From this base, rather complicated programming systems can be constructed while retaining much of the conceptual simplicity.

APPENDIX III

SUMMARY OF COMMAND ROUTINES